Programación avanzada en Python

Lección 4: Clases y objetos

eip ESCUELA INTERNACIONAL POSGRADOS

ÍNDICE

Clases y objetos

PRESENTACIÓN Y OBJETIVOS

En este cuarto capítulo vamos a adentrarnos en la programación orientada a objetos (POO). Estudiaremos sus y veremos los conceptos principales de este paradigma de programación (clases, objetos, atributos y métodos) altamente utilizado en Java o Python.

Objetivos

- En esta lección aprenderás a:
- Diferenciar entre POO y programación estructurada
- Trabajar con Clases, objetos y métodos.
- Fundamentos de POO

1. PROGRAMACIÓN ESTRUCTURADA FRENTE A POO

Un programa puede definirse con un conjunto de estructuras, módulos y funciones. Este tipo de programación es la programación estructurada.

En la programación estructurada existen estos 3 tipos de sentencias:

- Sentencias de selección

- Sentencias de iteración

- Sentencias de secuencia

Los programas están bien estructurados y procedimentales con estas sentencias.

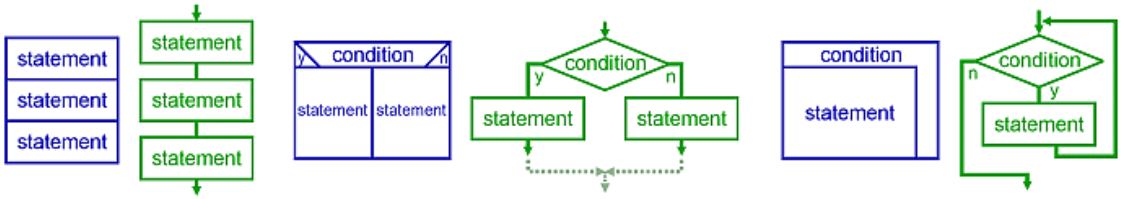

Figura 1.1 Ejemplo programación estructurada

En la programación orientada a objetos, los programas interactúan principalmente con objetos. Los objetos se basan en clases creadas en el programa. Los atributos y métodos también se crean en el programa y se realizan utilizando sus objetos de clase.

Hay 4 conceptos principales para la programación POO:

- Encapsulación
- Abstracción
- Polimorfismo
- Herencia

Volveremos a hablar sobre esto concepto más adelante

Principalmente, las clases contienen:

- Atributos
- Métodos

Las clases trabajan con objetos:

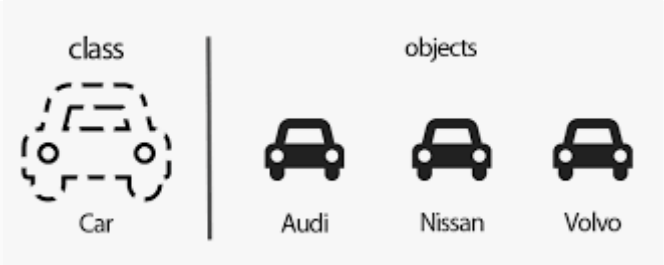

Figura 1.2 Ejemplo de programa OOP.

Programación estructurada

- El programa está dividido en un conjunto de subprograma denominados funciones o módulos
- La modificación de todo el contenido es difícil
- Las funciones interactúan en todo el programa pasando los valores por parámetros y devolviendo las salidas.
- Las funciones son generales, no se utilizan especificadores de acceso.
- Los datos no son seguros
- Código difícil de reutilizar
- Enfoque descendente
- Diseño del programa basado en el proceso
- No se permite la sobrecarga de datos
- Menos flexible
- Menor abstracción
- Lenguajes: C, basic, fortran, pascal

Programación orientada a objetos

- El programa contiene objetos de clases con atributos y métodos
- Es fácil modificar el programa y fácil hacer cambios a través de los objetos de las clases
- La interacción del programa se realiza principalmente con objetos y los valores son pasados por mensajes a través de objetos de clases.
- En las clases se utilizan especificadores de acceso como public, private y protected.
- Datos seguros
- El código es fácil de reutilizar
- Enfoque ascendente
- Diseño del programa basado en los datos
- Se permite la sobrecarga de datos
- Más flexible
- Mayor abstracción
- Lenguajes: python, php, C#, C++, java, ruby, java script

2. CLASES Y OBJETOS

Una clase contiene atributos y métodos. Un objeto es formado mediante la clase, los atributos y métodos.

En la figura anterior, el coche es la clase y los objetos son Audi, Nissan y Volvo. Podemos crear tantos objetos como queramos. Cada objeto es un coche, lo que significa que cada modelo es una clase de coche.

La clase es creada con la palabra clave 'class'

Sintaxis:

class class_name:

statements

Para crear objetos:

obj1 = class_name()

obj2 = class_name()

...

No hay límite en el número de objetos. Las clases contiene atributos y métodos como veremos a continuación.

3. ATRIBUTOS Y MÉTODOS

Dentro de la clase, hay atributos y métodos como podemos ver en la siguiente imagen.

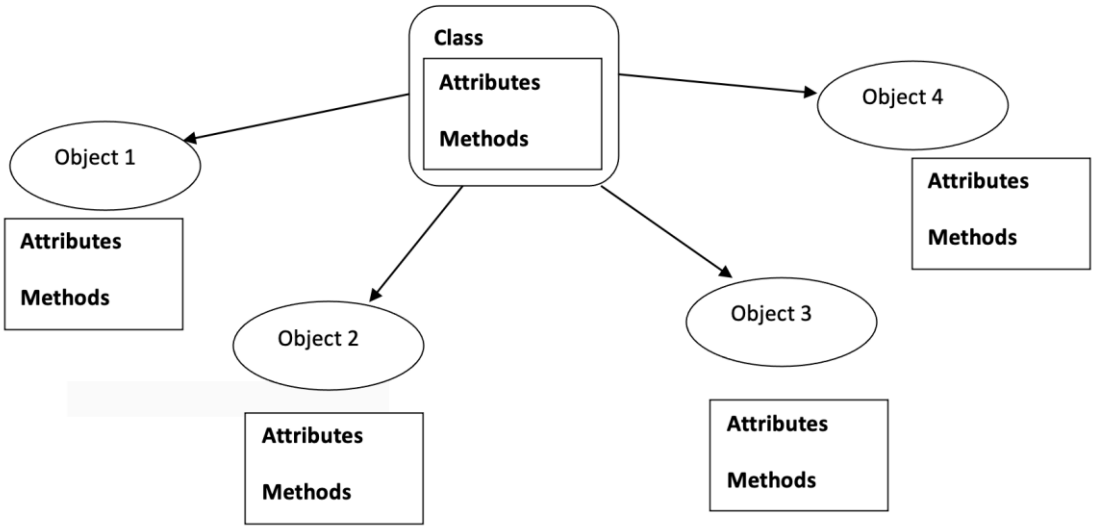

Figura 2.1 Atributos y métodos.

Atributos - tipos de datos variables que se definen e inicializan con valores.

Métodos - funciones que se definen y realizan el cálculo con los atributos especificados en la clase

Sintaxis:

```
class class_name:
        attribute1
        attribute2
        ..........
        def  method1():
                statements
                return
        ..........
```

Para crear un objeto con la clase:

Obj_name = class_name()

El objeto puede acceder a los atributos y métodos de la siguiente manera:

Obj_name.attribute1

Obj_name.attribute2

Obj_name.method1 ()

Veamos un ejemplo:

```
class my_car:
  new_car = "Ford"
```

Se trata de una clase llamada 'my_car' con el atributo 'new_car' y un valor 'Ford' asignado.

```
class my_car:
  new_car = "Ford"
```

```
car_1 = my_car()
```

car_1 es un objeto de la clase my_car

```
print(car_1.new_car)
```

```
Ford
```

Una clase no puede estar vacía. Se puede hacer vacía con la palabra clave pass as:

```
class myclass:
  pass
```

No devolverá ningún error y también la clase puede estar vacía.

4. MÉTODOS ESPECIALES

Las clases definen métodos que utilizan los atributos definidos en la misma clase:

```
class my_car:
  new_car = "Ford"

  def car_name(self):
    print(new_car)
```

Falta self. para poder hacer la llam

En la clase my_car, new_car es el atributo y car_name es el método que utiliza el atributo new_car.

Para crear el objeto:

```
obj = my_car()
```

Y llamando al método dentro de la función:

```
obj.car_name()
```

Al ejecutar esto, muestra un error

```
      5   def car_name(self):
----> 6     print(new_car)
      7
      8 # when the function call using object,

NameError: name 'new_car' is not defined
```

El error es new_car, el atributo no está definido incluso aunque el atributo se inicialice en la misma clase.

Entonces, ¿Por qué este error?

La función llama usando el objeto, pasando el objeto como parámetro a la función. Así que se debería llamar a los atributos dentro de la clase como 'obj.attribute_name' al llamar a los atributos de los métodos de la clase.

En la función definida, el objeto pasa como parámetro a self y el atributo es new_car. La llamada dentro del método sería:

self.new_car

```
class my_car:
  new_car = "Ford"

  def car_name(self):
    print(self.new_car)
```

Llamando a la función con el objeto:

```
obj = my_car()
```

```
obj.car_name()
```

```
Ford
```

Método init

- El método init es un método especial que comienza y termina con un doble guión bajo como __init__.
- Es el primer método dentro de la clase
- Funciona como constructor de la clase.
- Cuando el objeto de la clase es llamado automáticamente ejecuta la función init.

Normalmente en la función init los atributos pueden ser inicializados y los parámetros pueden ser pasados al método init.

El primer parámetro de la función __init__ suele ser self que pasa el objeto. Otros parámetros pueden ser definidos después de self según los requerimientos.

Cuando se crea el objeto, el método init se ejecuta automáticamente y los parámetros se pueden pasar al método init. Loos parámetros pueden ser dados a la clase en el momento de la creación del objeto.

Sintaxis:

clase nombre_clase:

    def __init__(self, param1,param2..):

        self.para1 = param1

        self.para2 = param2

        .....

El objeto de la clase se crea con los parámetros del método init como:

Objeto = nombre_de_la_clase (param1, param2...)

También se pueden crear otros objetos con diferentes parámetros.

Objeto1 = nombre_de_la_clase (paramA, paramB...)

Veamos un ejemplo:

```
class person:

  def __init__(self,name,age):
    self.name = name
    self.age = age

  def output_display(self):
    print("Name :" , self.name)
    print("Age :", self.age)
```

El método init es definido primero con 2 parámetros name y age junto con el parámetro por defecto self.

Los parámetros se inicializan dentro del método init con el objeto pasado a self y estos atributos se usan en el método output_display para simplemente imprimir los atributos pasados.

El objeto creado con los parámetros se haría de la siguiente manera:

```
person1 = person('x',55)
```

El método init se ejecuta y los parámetros son pasados a la función init, self.name y self.age se asignan con x y 55 respectivamente.

Si llamamos al método output_display, nos mostrará lo siguiente:

```
person1.output_display()
```

```
Name : x
Age : 55
```

Los atributos nombre y edad pueden ser llamados directamente por el objeto y el valor de los atributos puede modificarse especificándolo de la siguiente manera:

person1.age = 60

Los atributos pueden ser borrados con la clave 'del':

del person1.age

Variables de clase

Hay dos tipos de variables o atributos que se pueden definir dentro de cada clase, las variables de instancia y las variables de clase.

- **Variables de instancia**
  Las variables de instancia se definen dentro del método init y son únicas para cada instancia, no son compartidas para todas las instancias u objetos. Por ejemplo, en la clase anterior, cuando se crea cada objeto, los atributos nombre y edad se inicializan para cada objeto. Así que son únicas para todos los objetos y se llamadas como variables de instancia de esa clase.
- **Variables de clase**
  Las variables de clase son compartidas por cada objeto. Se definen en la clase y fuera de cada función. Cada objeto puede acceder a ellas a través del nombre del objeto o con el nombre de la clase
  Las variables de clase pueden simplemente asignar un valor a una variable fuera del método init.

  Veamos un ejemplo:

```
class Car:

    vechicle = 'car'

    def __init__(self, model, color):
        self.model = model
        self.color = color
```

En la clase car, vehicle = 'car' es una variable de clase que se define dentro de la clase y fuera de cada función. Así que es común para cada objeto de la clase.

Las variables model y color son variables de instancia que son únicas para cada objeto cuando se crea el objeto.

```
audi = Car("Audi", "white")
volvo = Car("Volvo","black")
```

La variable de instancia de audi será:

```
print(audi.model)
print(audi.color)
```

```
Audi
white
```

La variable de instancia de volvo será:

```
print(volvo.model)
print(volvo.color)
```

```
Volvo
black
```

Las variables de instancia de ambos objetos son únicas para cada uno. Comprobemos la variable de clase vehículo de estos 2 diferentes objetos

```
print(audi.vechicle)
print(volvo.vechicle)
```

```
car
car
```

Ambas son iguales, lo que significa que pueden ser compartidas por todos los objetos. La variable de clase también se puede acceder por el nombre de la clase coche como:

```
print(Car.vechicle)
```

```
car
```

5. HERENCIA

Uno de los conceptos importantes de la POO es la herencia. Si queremos construir una clase con las características de otra clase y queremos algunas características extras, la clase puede usar las propiedades de la clase original sin reconstruirla de nuevo.

Por ejemplo, tenemos las clases A y B.

A construye algunos atributos y métodos.

B necesita los mismos atributos y métodos desarrollados en la clase A y también necesita algunas propiedades adicionales.

En tal caso no es necesario volver a construir las propiedades de A en B. B puede usar las propiedades de A heredándolas de A. Esto se llama **herencia**. La clase A es llamada clase base y B se llama clase derivada, que hereda las propiedades de la clase base.

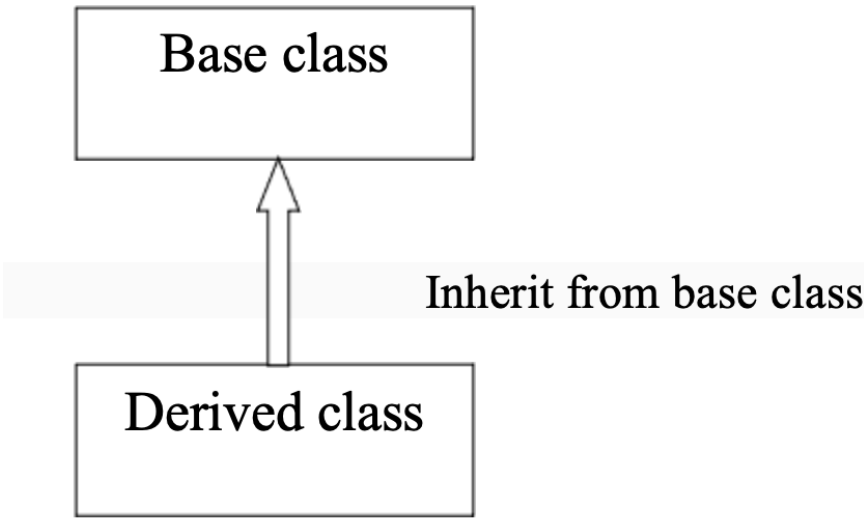

Figura 5.1 Herencia.

La clase base se llama clase padre o superclase y la clase derivada se llama clase hija o subclase.

Los códigos pueden ser fácilmente reutilizables usando las propiedades herencia de la clase base/superclase/padre a la clase derivada/subclase/clase hija.

La clase derivada puede heredar las propiedades especificando la clase base como parámetro.

Sintaxis:

Class base_class ():

    attribute = value

    def method():

        Statements

Class derived_class (base_class):

    Statements

Cuando se crea un objeto de la clase derivada, los métodos y atributos de la clase base pueden ser accedidos a través del objeto de la clase derivada.

Object = derived_class ()

Object.attribute

Object.method ()

Veamos un ejemplo:

```
class base_class():
  def base_method(self):
    print("I am from base class")
```

Esta es la clase base con un método definido en ella.

```
class derived_class(base_class):
  def derive_method(self):
    print("I am from derived class")
```

Esta es la clase derivada hereda la propiedad de la clase base dando la clase base como parámetro y finalmente define un método en ella.

Ahora se crea el objeto de la clase derivada.

```
object1 = derived_class()
object1.derive_method()
I am from derived class
object1.base_method()
I am from base class
```

Debido a que hereda de la clase base los métodos y atributos, estos pueden ser accedidos por su clase derivada.

Múltiples herencias

Una clase derivada puede heredar las propiedades de varias clases base. La clase derivada podrá acceder a los atributos y métodos de las clases base.

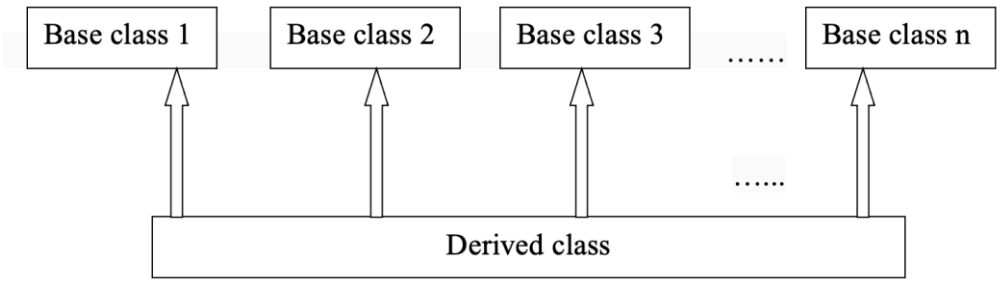

Figura 5.2 Clase derivada

Sintaxis:

Class base_class1():

    Statements

Class base_class2():

    Statements

.....

Class base_class n():

    Statements

Class derived_class (base_class1, base_class2, base_class3,...base_class n):

    Statements

Ejemplo:

```
class addition():
  def add(self,x,y):
    return x+y

class subtraction():
  def sub(self,x,y):
    return x-y

class calcualte(addition,subtraction):
  def mul(self,x,y):
    return x*y
```

Las clases addition y subtraction son clases base y calculate la clase derivada que hereda de las demás clases .

```
result = calcualte()
```

```
print(result.add(3,2))
print(result.sub(3,2))
print(result.mul(3,2))
```

```
5
1
6
```

Herencia de varios niveles

Una clase derivada puede heredar las propiedades de otra clase derivada y así sucesivamente. No hay límite en el número de clases derivadas que pueden heredar de otra.

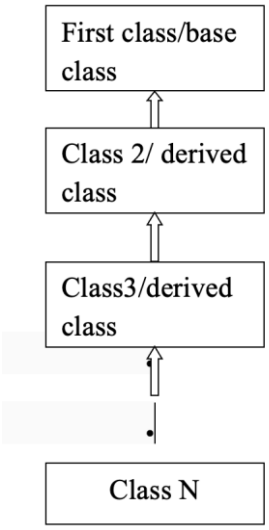

Figura 5.3 Herencia a varios niveles

Sintaxis:

Clase class1():

Statements

Clase class2(class1):

Statements

Clase n(clase n-1):

Statements

.....

Ejemplo:

```
class speaking():
  def speak(self):
    print("I am speaking")

class sleeping(speaking):
  def sleep(self):
    print("I am sleeping")

class eating(sleeping):
  def eat(self):
    print("I an eating")
```

A continuación, creamos un objeto y vamos llamando a las diferentes funciones.

```
me = eating()
```

```
me.eat()
```

```
I an eating
```

```
me.sleep()
```

```
I am sleeping
```

```
me.speak()
```

```
I am speaking
```

Sobreescritura de métodos

Normalmente heredamos los métodos de la clase base a la clase derivada. En algunos casos, debe haber algunas modificaciones o cambios necesarios para los métodos de la clase base.

La clase derivada puede definir de nuevo la misma clase con modificaciones. A esto se le llama sobreescritura de métodos.

Consideremos un ejemplo,

```
class speaking():
  def speak(self):
    print("I am speaking")

class sleeping(speaking):
  def speak(self):
    print("I am sleeping")
```

Speaking es la clase base y *speak* es el método de la clase base. La clase *sleeping* es la clase derivada, que hereda la propiedad de la clase base y de nuevo define el mismo método *speak* que el definido en la clase base.

Veamos que sucede cuando llamamos al método.

```
object = sleeping()
```

```
object.speak()
```

```
I am sleeping
```

En el anterior ejemplo, se imprime el mensaje que se define en la clase derivada. El método está sobrescribiendo el contenido del método speak.

Método issubclass

El método issubclass es una función incorporada que se utiliza para comprobar las relaciones entre las clases.

Sintaxis:

issubclass(clase_base, clase_derivada)

Devuelve un valor booleano, verdadero si la relación es correcta, en caso contrario devuelve falso. Veamos un ejemplo:

Las clases *addition y subtraction* son clases base. *Calculate* es su clase derivada.

```
class addition():
  def add(self,x,y):
    return x+y

class subtraction():
  def sub(self,x,y):
    return x-y

class calcualte(addition,subtraction):
  def mul(self,x,y):
    return x*y
```

```
print(issubclass(calcualte,addition))
print(issubclass(addition,subtraction))
```

```
True
False
```

6. ENCAPSULACIÓN

Uno de los conceptos de la POO es la encapsulación. Como su nombre indica, encapsula una clase, lo que significa que encapsula los atributos y métodos y oculta los datos al exterior.

Tomemos un ejemplo real, consideremos las clases de una escuela. Hay muchas aulas y cada aula tiene sus propios datos. De este modo, los datos de cada aula sólo están disponibles o accesibles por el profesor de su aula.

Por ejemplo, los datos de la 10ª aula sólo son accesibles por el profesor de la 10ª aula, cualquier otro profesor no puede acceder a ellos.

En el ejemplo de las aulas, los datos serán métodos y variables y tomarán cada clase como una aula.

Por ello, los métodos y variables solo son accesibles por su propia clase. Ninguna otra clase puede acceder a ella. Es oculto y seguro.

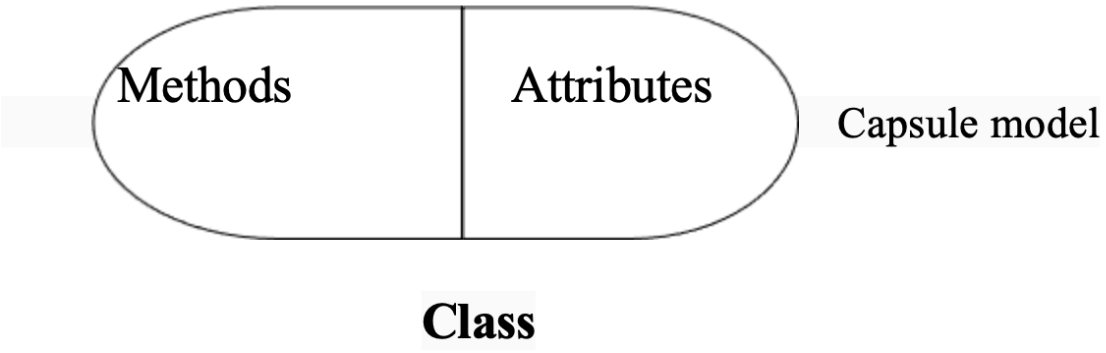

Figura 6.1 Modelo encapsulación

En Python los datos se encapsulan u ocultan usando los modificadores de acceso. Hay 3 tipos de modificadores de acceso:

- *Públic*
- *Private*
- *Protec*

En el encapsulamiento, sólo se utilizan los modificadores *private* y *protect* para que los datos sólo sean accesibles por su clase

Miembros protegidos

Los miembros protegidos comienzan con un guion bajo, '_'.

Sintaxis:

self._variable = value

Veamos un ejemplo:

```
class base_class:

    def __init__(self):
         self._protect = 2
```

Es una clase base con una variable protegida *protect* con valor 2. A continuación, se deriva otra clase de la clase base y se accede a la variable *protect* de la siguiente manera:

```
class derived_class(base_class):

    def __init__(self):

        # Calling constructor of Base class
        base_class.__init__(self)
        print("Calling protected member of base class: ")
        print(self._protect)
```

Resultado:

```
object1 = derived_class()
```

```
Calling protected member of base class:
2
```

La variable protegida es accedida por su propia clase derivada. Si llamamos directamente a la variable protect fuera de la clase:

```
object2.protect
```

Muestra un error porque se ha definido como protegida:

```
----> 1 object2.protect

AttributeError: 'base_class' object has no attribute 'protect'
```

Miembros privados

Los miembros privados comienzan con un doble guion bajo, '__'. Los miembros privados no son accedidos desde fuera de la clase ni por una clase derivada.

Veamos un ejemplo:

```
class base_class:

    def __init__(self):
         self.__protect = 2
```

La variable protegida se define como privada, cuando una clase derivada llama al miembro privado ocurre lo siguiente:

```
class derived_class(base_class):

    def __init__(self):

        # Calling constructor of Base class
        base_class.__init__(self)
        print("Calling protected member of base class: ")
        print(self.__protect)
```

Resultado:

```
object1 = derived_class()
```

```
----> 8         print(self.__protect)

AttributeError: 'derived_class' object has no attribute '_derived_class__protect'
```

La encapsulación es una capa protectora para los métodos y variables de una clase.

7. POLIMORFISMO

Polimorfismo significa diferentes formas en distintas condiciones. En el entorno de Python significa que las propiedades como métodos y entidades muestran diferentes formas o resultados en diferentes tipos.

Veamos un ejemplo:

Consideremos una operación de suma

```
a = 100
b = 3.4333
sum = a + b
print(sum)
```

```
103.4333
```

Aquí los dos números se suman con el operador '+'. Consideremos también la concatenación de cadenas, utilizando el operador '+' las dos cadenas pueden ser concatenadas.

```
str1 = "hello "
str2 = "xyz"
str3 = str1 + str2

print("String after concatenation:", str3)
```

Se concatenan las cadenas 1 y 2 y se almacenan en la cadena 3

```
String after concatenation: hello xyz
```

El operador '+' es el mejor ejemplo de polimorfismo, muestra diferentes formas del signo '+'.

Veamos otro ejemplo con el atributo len (), que se utiliza para calcular la longitud

```
print(len("string"))
print(len(['list_item1','list_item2']))
print(len({'key1':'value1','key2':'value2'}))
```

```
6
2
2
```

- Para una cadena, len () calcula el número de caracteres
- Para una lista, len () calcula el número de elementos
- Para un diccionario, len () calcula el número de pares clave-valor

Usando el método len () se puede calcular el número de diferentes tipos de formas.

Consideremos las clases:

```
class speaking():
  def speak(self):
    print("I am speaking")

class sleeping():
  def speak(self):
    print("He also speaking")
```

Speaking y sleeping son dos clases diferentes con un atributo con el mismo nombre. Cuando iteramos a través de la tupla de objetos ocurre lo siguiente:

```
obj1 = speaking()
obj2 = sleeping()
```

```
for objects in (obj1,obj2):
  objects.speak()
```

```
I am speaking
He also speaking
```

La variable 'objects' recorre cada objeto y ejecuta las funciones sin considerar su tipo.

8. ABSTRACCIÓN

La abstracción es otro de los conceptos de la POO y se utiliza para ocultar los datos del exterior. Los datos se abstraen usando el modificador de acceso private.

Los atributos que comienzan con doble guion bajo '__' no pueden ser accedidos desde el exterior de la clase.

Veamos un ejemplo:

Creamos una clase *abstract* con el atributo privado *count* definido como __count, por lo que, está oculto desde fuera de la clase.

```
class abstract:
    __count = 0;
    def __init__(self):
        abstract.__count = abstract.__count+1
    def display(self):
        print("The number of employees",abstract.__count)
```

A continuación, creamos dos objetos que llaman a la función init e incrementa el valor de *count* a 2

```
obj1 = abstract()
obj2 = abstract()
```

Así que, si se llama a la función display, muestra el siguiente resultado

```
    obj1.display()
```

```
The number of employees 2
```

En cambio, si se llama al atributo se muestra un error, porque es privado para la clase y no se puede acceder desde fuera de la clase.

```
print(obj1.__count)
```

```
AttributeError: 'abstract' object has no attribute '__count'
```

9. PUNTOS CLAVE

- La **programación estructurada:** Un programa puede definirse con un conjunto de estructuras, módulos y funciones.
  En la **programación orientada a objetos,** los programas interactúan principalmente con objetos. Los objetos se basan en clases creadas en el programa.
- Una **clase** contiene **atributos** y **métodos**. Un objeto es formado mediante la clase, los atributos y métodos.
- La **herencia** nos permite construir una clase con las características de otra clase.
- La **encapsulación** es una capa protectora para los métodos y variables de una clase.
- El **polimorfismo** significa que las propiedades como métodos y entidades muestran diferentes formas o resultados en diferentes tipos.
- La **abstracción** se utiliza para ocultar los datos del exterior. Los datos se abstraen usando el modificador de acceso *private*.